LANGLEY GRANT

IN-61-CR
20604
P56

An Annual Progress Report
Grant No. NAG-1-1123

April 1, 1990 - March 31, 1991

SOFTWARE FAULT TOLERANCE USING DATA DIVERSITY

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention:

Dr. D. E. Eckhardt, Jr., ISD
M/S 478

Submitted by:

John C. Knight
Associate Professor

Report No. UVA/528344/CS92/101
July 1991

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF
ENGINEERING
& APPLIED SCIENCE

University of Virginia
Thornton Hall
Charlottesville, VA 22903

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 600. There are 160 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Applied Mechanics, Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 17,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.

An Annual Progress Report
Grant No. NAG-1-1123

April 1, 1990 - March 31, 1991

SOFTWARE FAULT TOLERANCE USING DATA DIVERSITY

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention:

Dr. D. E. Eckhardt, Jr., ISD
M/S 478

Submitted by:

John C. Knight
Associate Professor

Department of Computer Sciencew
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528344/CS92/101
July 1991

Copy No.____

TABLE OF CONTENTS

1. INTRODUCTION

During the grant reporting period, the research carried out has involved the development of the technique known as *data diversity*. Data diversity relies on a different form of redundancy from existing approaches to software fault tolerance and is substantially less expensive to implement. Data diversity can also be applied to software testing and greatly facilitates the automation of testing. Up to now, it had been explored both theoretically and in a pilot study, and had been shown to be a promising technique. We have continued to develop data diversity in various areas under this grant. The basic concepts of data diversity and its relation to other techniques are presented in section 1 of this report. Previous work and basic concepts are reviewed in section 2. The results obtained under this grant are presented in two self-contained papers that are included in this report as appendices.

Various methods for making software that is fault-tolerant have been proposed in an effort to provide substantial improvements in the reliability of software for safety-critical applications. At execution time, the fault-tolerant structure attempts to cope with the effect of those faults that have survived the development process. The two best-known methods of building fault-tolerant software are *N*-version programming [3] and recovery blocks [7]. To tolerate faults, both of these techniques rely on *design diversity*, i.e., the availability of multiple implementations of a specification. Software engineers assume that the different implementations use different designs and thereby, it is hoped, contain different faults.

N-version programming requires the separate, independent preparation of multiple (i.e. "*N*") versions of a program for some application. These versions are usually executed in parallel in the application environment; each receives identical inputs, and each produces its version of the required outputs. A decision function collects the outputs and selects the system's output from them.

The recovery block structure submits the results of an algorithm to an acceptance test. If the results fail the test, the system restores the state of the machine that existed just prior to execution of the algorithm and executes an alternate algorithm. The system repeats this process until it exhausts the set of alternates or produces a satisfactory output.

It is well known that software often fails for special cases in the data space†. In practice, a program may survive extensive testing, work for many cases, and then fail on a special case. The special case may take the form of what seems to be an obscure set of values in the data. Testing frequently fails to reveal faults associated with special cases precisely because the test harness does not generate the exact circumstances required. A test data set whose values are merely close to the values that cause the program to fail does not uncover the fault.

These observations suggest that if software fails under a particular set of execution conditions, a minor perturbation of those execution conditions might allow the software to work. Other researchers have exploited this property in specific instances. Gray observed that certain faults that caused failure in an asynchronous commercial system did not always cause failure if the same inputs were submitted to a second execution [5]. The system succeeded on the second execution due to a chance reordering of the asynchronous events. Gray introduced the term "Heisenbugs" to describe these faults and their apparent non-deterministic manifestations.

Morris has proposed "temporal separation" of the input data to a dual version system [6]. The versions use data from adjacent real-time frames rather than the same frame. Since the versions read sensor data at different times, the data tend to differ. The system corrects for this discrepancy so that it can vote on the outputs of the versions. It is hoped that the use of time-skewed data will prevent the versions from failing simultaneously.

†For example, see [8], pp. 347-348.

Each of these approaches attempts to avoid faults by operating software with altered execution conditions. Each approach relies upon circumstance to change the conditions. However, execution conditions can be changed deliberately. For example, concurrent systems need not rely on a chance reordering of events. If reordering events might allow a second execution to succeed, then the system should enforce a reordering. Changing the processor dispatching algorithm after state restoration forces a different execution sequence. Similarly, skewing the inputs to the versions in an *N*-version system does not require the passage of time. Inputs can be manipulated algorithmically. Many real-valued quantities have tolerances set by their specifications, and all values within those tolerances are logically equivalent.

Data diversity is an orthogonal approach to design diversity and a generalization of the work cited above. A diverse-data system exploits the fact that a slight change in the operating conditions, particularly the input data, is often sufficient to permit a program to execute correctly.

2. PREVIOUS WORK

Previous work has concentrated on completing the definition of the concepts of data diversity, developing preliminary performance models, and performing a pilot performance study. In this section we review the fundamental concepts associated with the technique. More details can be found in [1, 2].

2.1. Failure Regions

The input data for most programs comes from hyperspaces of high dimension. For example, a program may read and process a set of twenty floating-point numbers, and so its input space has twenty dimensions. In many cases the number of dimensions in the space varies dynamically because the amount of data that a program processes varies for different executions.

The *failure domain* of a program is the set of input points that cause program failure [4]. We call a failure domain along with its geometry a *failure region.* A failure region describes the distribution of points in the failure domain and determines the effectiveness of data diversity. The fault tolerance of a system employing data diversity depends upon the ability of the reexpression algorithm to produce data points that lie outside of a failure region, given an initial data point that lies within a failure region. The program executes correctly on reexpressed data points only if they lie outside a failure region. If the failure region has a small cross section in some dimension(s), then reexpression should have a high probability of translating the data point out of the failure region.

2.2. Data Reexpression

At its simplest, *data reexpression* is the generation of logically equivalent data sets. Figure 1 illustrates this basic form of data reexpression. An input x given directly to a program *Program* produces an output *Program*(x). Alternatively, a reexpression algorithm *Express* transforms the original input x to produce a new input y, where $y = Express(x)$. The input y may contain exactly the same information as the input x, but in a different form, or y may approximate the information

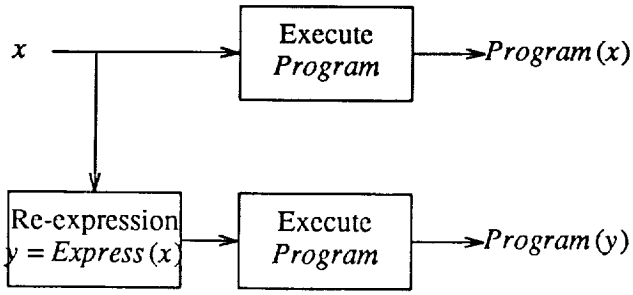

Figure 1. Re-expression.

in input x. The program *Program* operates on the reexpressed input y to produce *Program*(y). *Program* and *Express* determine the relationship between *Program*(x) and *Program*(y). Data diversity can tolerate faults when *Program*(y) is a useful output but *Program*(x) is not.

Figure 2 identifies two classes of reexpression algorithms. The first class produces elements in the set I, i.e., elements that should produce *I*dentical outputs, up to numerical error, when processed by the program. These algorithms are termed *exact*. The second class yields elements in the set V, i.e., elements that should produce *V*alid but not necessarily identical outputs when processed by the program. These algorithms are termed *approximate*. Exact algorithms are desirable from the viewpoint of error detection since they permit error detection by comparison of outputs that should be the same. Approximate algorithms might be easier to produce, however, and might have a better chance of allowing the program to escape a failure region. Exact reexpression algorithms might also have the defect of preserving precisely those aspects of the data that cause failure.

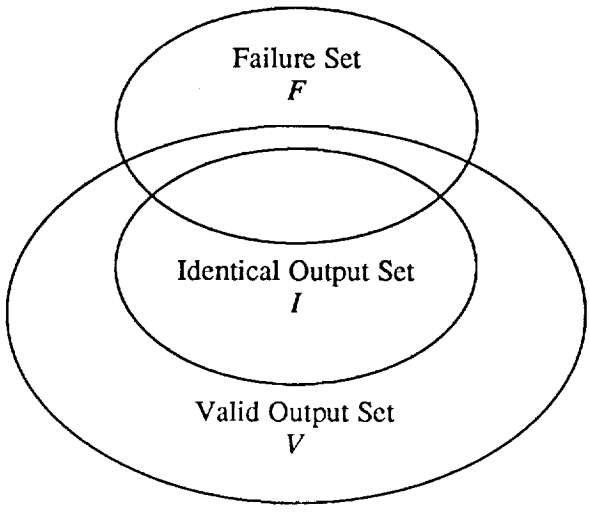

Figure 2. Sets in the Output Space for Given Input x.

As an example of an exact reexpression algorithm, consider a program that processes inputs representing Cartesian coordinates and that only the points' relative positions are relevant to the application. A valid reexpression algorithm could translate the coordinate system to a new origin or rotate the coordinate system about an arbitrary point.

Any mapping of a program's data that preserves the information content of the data is a valid reexpression algorithm. A simple approximate data reexpression algorithm for a floating-point quantity might alter its value by a small percentage. The allowable percentage by which the data value could be altered would be determined by the application. In applications that process sensor data, for example, the accuracy of the data is often poor and deliberate small changes are unlikely to affect performance.

The reexpression algorithms discussed above depend on numeric manipulation. It might appear that many non-numeric applications are not candidates for fault tolerance through data diversity because numeric reexpression cannot be used. The source program input to a compiler, for example, is a character string, and any changes to that character string will almost certainly change the meaning of the program.

Data can take other forms, however, and data diversity can be applied successfully to other applications. Consider a compiler with a conventional multi-pass organization. Although the initial representation of a source program is a character string, the program may be represented in many ways during compilation, for example, as a tree. There are many transformations that can be applied to a tree that preserve semantics. A compiler employing data diversity for its later passes could be constructed by executing these later passes on several different copies of the tree obtained by semantics preserving transformations.

Similarly, the order of storage allocation in an activation record is usually determined by the programmer's order of declaration. In practice, this order need not be preserved, and a set of

semantically equivalent internal representations of a program can be obtained by shuffling the order of variables in activation records.

Combining tree transformations, data storage reordering, and code storage reordering, i.e., generation of code for subprograms in an arbitrary order, provides considerable diversity in the data processed by large fractions of a conventional compiler. These approaches to reexpression are exact in that, although the code generated by the compiler may be different and therefore not amenable to any simple selection algorithm, the effects of these programs should be identical, and so simple voting can be used for selection during execution of the programs generated by the compiler.

In general, a reexpression algorithm must be tailored to the application at hand. Producing a reexpression algorithm requires a careful analysis of the type and magnitude of reexpression appropriate for each candidate datum. Simpler reexpression algorithms are more desirable than complex ones since they are less likely to contain design flaws.

2.3. Retry Blocks

A *retry block* is a modification of the recovery block structure that uses data diversity instead of design diversity. The concept is illustrated in figure 3. Rather than the multiple alternate algorithms used in a recovery block, a retry block uses only one algorithm. A retry block's acceptance test has the same form and purpose as a recovery block's acceptance test. A retry block executes the single algorithm normally and evaluates the acceptance test. If the acceptance test passes, the retry block is complete. If the acceptance test fails, the algorithm executes again after the data has been reexpressed. The system repeats this process until it violates a deadline or produces a satisfactory output.

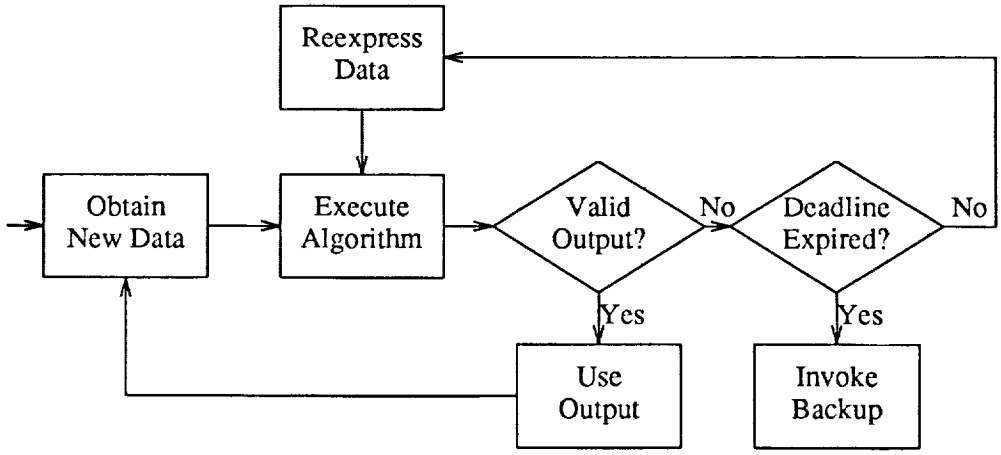

Figure 3: Retry Block.

2.4. *N*-Copy Programming

An *N-copy* system is similar to an *N*-version system but uses data diversity instead of design diversity. The concept is illustrated in figure 4. *N* copies of a program execute in parallel; each on a set of data produced by reexpression. The system selects the output to be used by an enhanced voting scheme.

Voting in an *N*-copy system is not necessarily straightforward. If the reexpression algorithm is exact, so that all copies should generate identical outputs, then a conventional majority vote can be used. However, if the reexpression algorithm is approximate, the copies could produce different but acceptable outputs. This is likely to occur near boundaries in the output space. In that case, a simple majority may not exist and the selection process is more involved. If a particular output occurs more than once, then it might be selected. If more than one output occurs more than once or no output occurs more than once, then selection might have to involve an arbitrary choice.

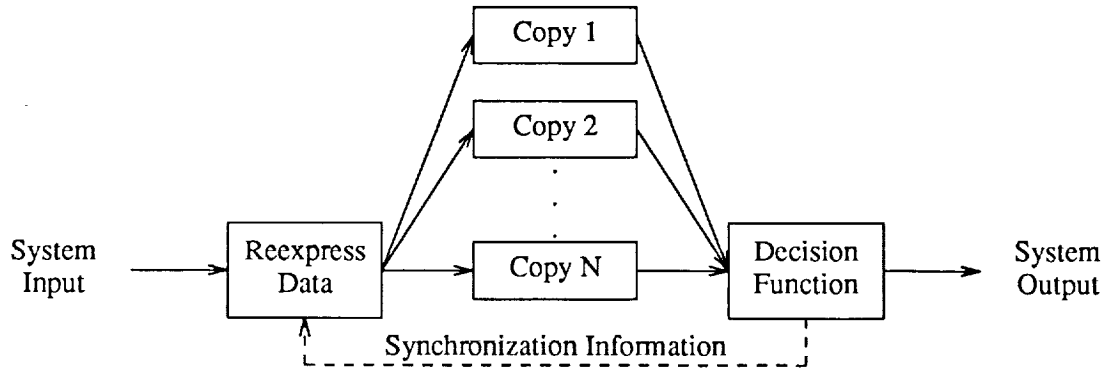

Figure 4: N-Copy Programming.

REFERENCES

[1] P.E. Ammann, "Data Diversity: An Approach to Software Fault Tolerance Ph.D. dissertation, Department of Computer Science, University of Virginia, February 1988.

[2] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance *IEEE Transactions on Computers*, Vol. 37, No. 4, April 1988.

[3] A. Avizienis, "The *N*-Version Approach to Fault-Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985.

[4] F. Cristian, "Exception Handling", in *Resilient Computing Systems, Volume 2*, T. Anderson, ed., John Wiley & Sons, New York, 1989.

[5] J. Gray, "Why do Computers Stop and What Can Be Done About It?", Tandem Technical Report 85.7, June 1985.

[6] M.A. Morris, "An Approach to the Design of Fault Tolerant Software", MSc Thesis, Cranfield Institute of Technology, September, 1981.

[7] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[8] "Tutorial: Software Testing and Validation Techniques", 2nd Ed., IEEE Computer Society Press, 1981.

APPENDIX A

On The Effectiveness Of Data Diversity

As An Error Detection Mechanism

ON THE EFFECTIVENESS OF DATA DIVERSITY AS AN ERROR DETECTION MECHANISM[†]

John C. Knight
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, Virginia, 22903
(804) 924-7605 FAX: (804) 982-2214
knight@virginia.edu

Paul E. Ammann
Department of Information Systems and Systems Engineering
George Mason University, Fairfax, VA 22030
(703) 764-4664 FAX: (703) 323-2630
pammann@gmuvax2.gmu.edu

Steven J. Santos
Department of Computer Science
University of Virginia, Charlottesville, Virginia, 22903
(804) 924-7605 FAX: (804) 982-2214
sjs2g@cs.virginia.edu

ABSTRACT

We have proposed previously an approach to software fault tolerance based on diversity in the data space; a technique we term *data diversity*. In a fault-tolerant system employing data diversity, the *same* software is executed on several set of points in the data space, each of which should produce the same (or simply related) output. Error detection in data diversity is achieved by observing unexpected differences between the outputs of the software on the different data points. This approach to error detection can be employed in software testing as well as fault tolerance, and an important question in both applications is the error detection performance of data diversity. We report in this paper on a study in which several pieces of commercial software were obtained and into which faults were injected. These faulty programs were tested using data diversity for error detection. The results of this study show that data diversity can be an effective error detection technique.

Subject Index:
Fault-tolerant software, software reliability, design diversity, data diversity.

[†]Sponsored in part by NASA grant NAG_1-1123-FDP.

1. INTRODUCTION

Error detection is the determination that the output of a software system is unacceptable to its application, i.e., the software is not in compliance with its specification. Clearly, error detection is a critical element of both software testing and software fault tolerance. In software testing, for example, if a test input has been created that causes a fault to manifest itself, this will be of little value unless the erroneous software operation can be detected. Similarly, it is not possible to mask the effects of a fault during execution, and thereby tolerate it, if the erroneous software operation is not detected. In some systems, detection of errors during operation rather than fault masking is the only goal [1].

Most existing approaches to software fault tolerance are based on the notion of *design diversity* [2]. These approaches employ some form of software redundancy, and depend for their operation upon differences in the various redundant designs. In the *N*-version approach to software fault tolerance, for example, independent implementations of the software are prepared [2, 3]. Although various error detection mechanisms are employed in such systems, an unexpected difference in the outputs of these different implementations is the primary mechanism for error detection.

In testing software systems, many different forms of error detection are used including replication checks, reversal checks, timing checks, and acceptability checks. The approach to error detection in which the output of a software system is desk checked by a human is actually a replication check. Similarly, human examination of the output to make sure that it "looks right" is actually an informal acceptability check. Many checking techniques are actually applications of design diversity.

We have proposed previously an approach to software fault tolerance based on diversity in the data space; a technique we term *data diversity* [4]. In contrast to systems employing design diversity, a system employing data diversity executes the *same* software on several *different* (but related) sets of points in the data space. If no fault

manifests itself, each input data set should produce the same (or simply related) output, and a decision algorithm is employed to determine system output. As with design diversity, it is an unexpected difference in the outputs that signals an error in a data-diverse system. Where there is such a difference, at least one of the executions of the software must have involved the manifestation of a fault. Error detection is followed in a fault-tolerant system by the selection of a suitable system output by a decision algorithm. In direct analogy to the *N*-version [3] and recovery block [5] strategies of design diversity, the decision algorithm in a data-diverse system uses a voter or an acceptance test. Data diversity was shown to provide a useful degree of software fault tolerance in a pilot study [6].

The fact that data diversity supports error detection is what permits its use in software fault tolerance. However, this ability to detect errors suggests that data diversity can be employed in software testing also. In practice, what is required in both cases is for errors to be detected with probability one when they occur and with probability zero otherwise. Not detecting an error when it occurs is potentially catastrophic in a fault-tolerant system. Signalling an error when none has occurred is also serious in a fault-tolerant system since this also could lead to failure. Not detecting an error that occurs when testing is less serious. Provided the fault responsible for the undetected error subsequently causes another error that *is* detected, the presence of the fault will be revealed. At test time, the important thing is that the errors produced by a given fault have non-zero probabilities of manifestation and of detection. The larger the probabilities the better, but non-zero probabilities ensure that the probability of finding the fault can be made as close to one as desired by increasing the number of tests. During testing, false error signals lead to wasted human effort.

The performance of data diversity as an aid to testing and as a technique for fault tolerance hinges on its error detection performance. As part of a long-term research project on software dependability, we are assessing the performance of various aspects of data diversity. In this paper we report on a study of its effectiveness in error detection.

We have performed an experiment in which the ability of a data-diverse system to detect the effects of injected faults was determined. In the next section, we describe the concepts of data diversity and the mechanisms of error detection in more detail. In section 3 we review the experiment that was performed and in section 4 we present the results of the experiment. Section 5 contains our conclusions.

2. ERROR DETECTION AND DATA DIVERSITY

The idea of using data diversity as a technique for achieving software fault tolerance was based on two observations. First, anecdotal evidence suggests that software tends to fail on special input cases and unusual combinations of events, and second, in many cases, software is a many-to-one mapping. Every programmer is familiar, for example, with the situation in which a program fails because of the incorrect processing of unexpected or obscure data cases. Similarly, programmers are aware that the exact order of the data read by a program is frequently irrelevant.

The first observation leads to the idea that software would not fail because of a specific special case if the special case were avoided, and the second observation suggests that the special case might be avoided by using a different input data set. Clearly, this different data has to be chosen so that an acceptable output is produced. A fault-tolerant software structure called an *N*-copy system [6] that is designed to take advantage of these ideas is shown in Figure 1.

The process of deriving input data sets that are equivalent to the original yet different in some way is termed *data reexpression*. There are two forms of data reexpression - *exact* and *approximate*. Exact data reexpression takes the supplied input data set and produces an alternative input data set that will produce either the same output as the original (perhaps up to the limits of numeric accuracy) or an output that is simply related to the original. An example of exact reexpression for a sort function is a random permutation of the input. Clearly the reexpressed input is different from the original input yet should produce identical output. Another example of exact

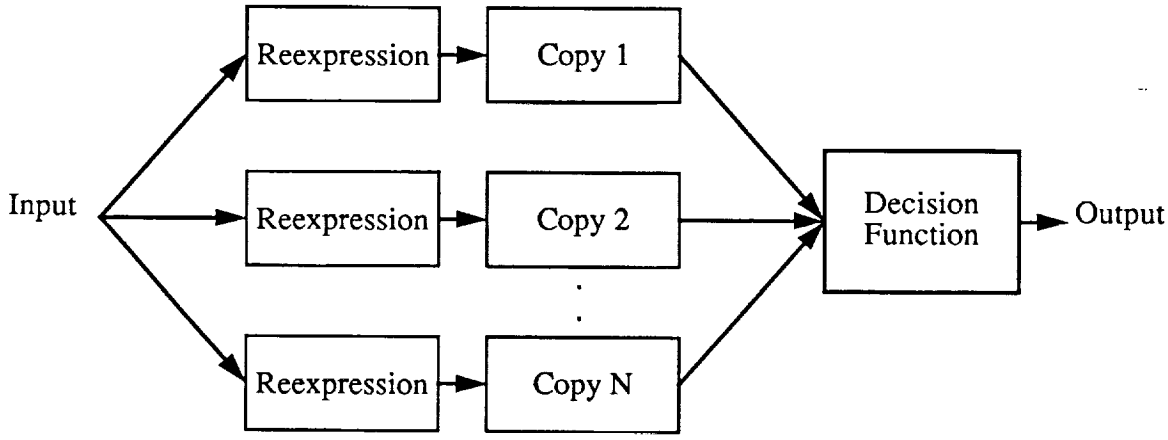

Figure 1 - *N*-Copy System Structure

reexpression for a sort function is to subtract each input data value from a value larger than all the input data values. Again, the reexpressed input data set is different from the original but the output will be different also. However, the output is simply related to the original and the original output is easily recovered.

Approximate data reexpression produces an alternative input data set that will produce an output acceptable to the application but possibly not the same nor simply related to that produced by the original data set. An example is the introduction of a low intensity noise term into the sensor values used by a control system. This should have little or no impact on the application since sensor data is usually of limited accuracy. However, the reexpressed data might avoid a failure arising from a special case in the data, such as a division by zero resulting from the subtraction of two identical data values.

Clearly, data reexpression algorithms are application-dependent. There is no general rule that permits reexpression algorithms to be derived for all applications, although this can be done in some special cases (see [7], for example). Our experience to date is that data reexpression algorithms exist for a surprisingly wide range of applications [8]. The efficacy of data reexpression is an important factor in the success of data diversity. It is data reexpression that yields the multiple data sets which ultimately lead to the unexpected difference in the outputs and thereby to error detection.

In summary, error detection using data diversity involves execution of the same software with different data sets where the different data sets should produce identical or related outputs. The different data sets are produced by deliberate manipulation in a process called reexpression. An error is signalled, and hence the manifestation of a fault observed, if there is an unexpected difference in the outputs from the various executions of the software. A potentially valuable benefit of detecting errors at test time in this way is that it permits automation of error detection in much the same way as comparison checking does [9, 10]. This facilitates automation of the whole test process. An automated test system employing data diversity is shown in Figure 2.

3. EXPERIMENTAL ASSESSMENT

3.1. Overview

The overall goal of this study was to assess the error detection performance of data diversity. Clearly, an analytic approach in which quantifiable characteristics of a software system are used to predict error detection performance would be ideal. However, such an approach is infeasible in general, and we resorted to an experimental approach in which we measured the ability of data diversity to detect errors in sample programs containing known faults. Informally, the approach was to execute samples of commercial software in an *N*-copy structure with input data taken from the appropriate operational distribution.

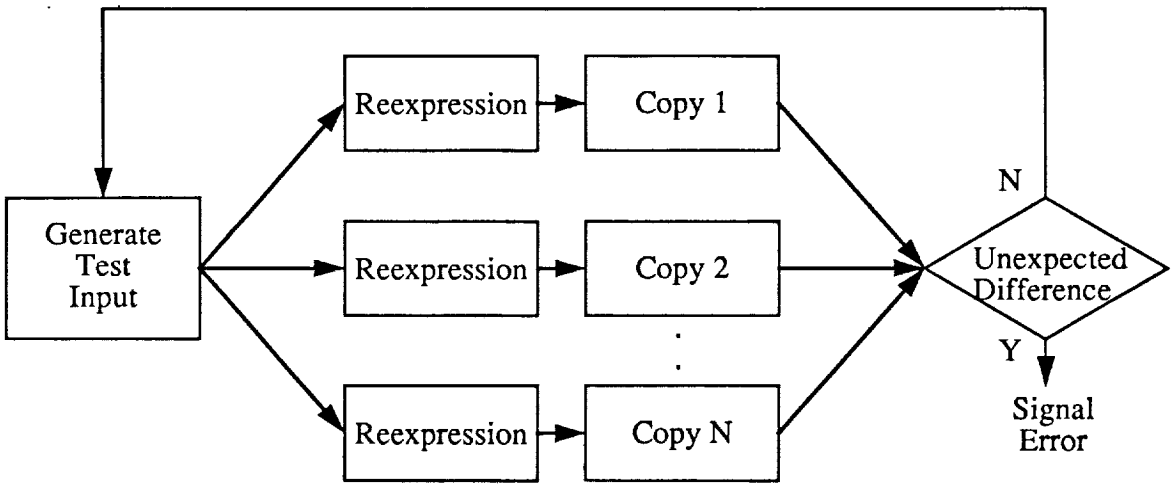

Figure 2 - Automated Test System Using Data Diversity

An experimental assessment of this type yields the most useful results when it is based on production software with faults left undetected by the development process. Statistically valid general results are obtained when large numbers of faults covering the known spectrum of fault types can be studied. Unless a large number of programs with a wide range of characteristics and a large number of faults of various types can be studied, the observed behavior has little statistically predictive meaning. Merely observing the performance of one or a small number of what would amount to randomly selected programs with essentially random faults does not necessarily lead to any useful general indication of the error detection performance of data diversity. Issues that would remain include the relevance of the results to other faults and fault types, the applicability of the results to software for different application areas, the effect of the skills of the programmers involved and their backgrounds, the effect of the specific data reexpression algorithm used, the effect of the software development environment, etc.

The size of the experimental study that the above implies is beyond our resources. Also, unfortunately† the production software that was available for study contained no known flaws. To mitigate the effect of the inevitable resource limitation and to obtain more than anecdotal evidence of performance, we chose to perform an experiment in which we constrained most of the independent variables in the experiment. That is, we fixed most of the factors that could influence the results and varied one factor completely and another somewhat. Thus our results are conditional on the values of the fixed factors but complete for one variable and indicative for the other. Specifically, we focused on a small number of samples of software written in C that were written by the same programming organization and that come from the same application domain. The domain used was geometric calculations associated with maritime navigation.

The study was performed in two phases. In the first phase, we explored the effects of a set of different fault types using fault injection. This study was complete in that all possible faults of certain types were generated and the resulting programs tested. The goal was to acquire information about the performance of data diversity for all possible instances of a set of fault types in a program. For example, each arithmetic operator in a given test program was systematically changed to every other possible arithmetic operator. This was repeated for all the arithmetic operators in the program and the resulting series of programs containing injected faults, or mutants [11], were tested. The same procedure was followed in phase one for all the relational operators and all the logical operators in the available test programs. Since all the arithmetic, logical, and relational operators were changed and each was changed into all possible alternative operators, this phase of the experiment provides a general indication of the ability of data diversity to detect faults characterized by the incorrect use of these operator types.

This kind of systematic fault injection can only be done for small test programs since the resulting number of mutant programs is very large and grows rapidly with the size of the initial test program. We generated all possible mutants in certain classes for

† Or fortunately, depending on your viewpoint.

two small programs. Each mutant was executed with realistic inputs using data diversity for error detection. Some of the mutants were benign, i.e., the change to the original program did not cause a fault, and others did not fail on every test case. The original software was used as an oracle to detect the cases where an error occurred, and hence when data diversity should have detected the error.

In the second phase of the study, we sought information on the effect of program size. The central question was whether the performance observed in the first phase was likely to scale up to larger programs. Again, resources were limited and only a single larger program was studied. For that program, we generated a random sample of the mutants since generating and testing all possible mutants was infeasible.

In both phases of the study each mutant was tested many times but the entire test process itself was also repeated many times. This was to obtain some indication of the variance in the observed performance. Testing a mutant with a single set of test cases reveals a certain performance level but this observation is a random sample subject to statistical variance. Repeating the test process many times with different test data each time gave insight into this variance which proved to be quite wide.

3.2. Subject Programs

The programs used in this study were supplied by Sperry Marine, Inc., of Charlottesville, Va. Sperry Marine manufactures computerized maritime electronic systems. The two programs used in the first phase of the study compute heading and distance from a ship's present position to its desired destination. Both present position and desired destination are supplied as latitude/longitude pairs. The outputs are the heading that must be steered and the distance to the destination in nautical miles. The first of the two test programs computes the great-circle course and the second computes the rhumb-line course. The great-circle course is that which minimizes the distance traveled between two points. Following a great-circle route requires constant adjustment of the heading. The rhumb-line course is the course that maintains a constant heading.

Although the traveled distance is longer, the ability to steer a fixed heading is a considerable benefit.

The second phase of the study used a program that computes the orbital position of a satellite in the Global Positioning Satellite System (GPSS). The GPSS is used in navigation systems to determine the location of an observer. The orbital position that is computed is the location of the satellite at some specified time in a Euclidean frame of reference based on the center of the Earth. The computations are based on orbital parameter information for a particular reference time that includes the inclination of the orbit above the equatorial plane, the orientation of the orbit relative to the zero meridian, and a large number of data elements used in correcting anomalies to increase the accuracy of the final position estimate.

3.3. Reexpression Algorithms

In phase one, three reexpression algorithms were used. The first depends upon the fact that the output of the two programs was a course heading and distance with no dependence on the actual values of latitude and longitude that were input. The reexpression algorithm rotated the input points around the Earth by an arbitrary amount. Essentially this amounts to changing the longitude but fixing the latitude. Clearly, this should have no effect on the output and so this is an exact reexpression algorithm.

The second reexpression algorithm was approximate. The supplied latitude/longitude pairs were modified by adding a small random offset to each. The resulting courses and headings produced should be very similar, and this required similarity was used as the error indicator.

The third reexpression algorithm was also approximate. It took the supplied latitude/longitude pairs and generated inputs that should produce almost identical outputs by shifting each location by one degree latitude East and one degree latitude West. These reexpressed locations along with the original data produce a total of six different

required courses, each of which should have very similar headings and distances. By arranging the data sets in an appropriate way, the output was forced to be ordered and this required ordering was used as the error indicator.

In the second phase of the experiment, four reexpression algorithms were used. Three of the four were exact and one was approximate. In the first exact reexpression algorithm the inclination of the satellite's orbit with respect to the equatorial plane (an input) was reversed. The effect of this reexpression on the output should be to reverse the sign of the output z coordinate. The second and third reexpressions were similar but involved the orientation of the orbit to the zero meridian (another input). Finally, the approximate reexpression algorithm perturbed the inclination of the orbit and its orientation to the zero meridian by a small random amount. The effects of these changes on the individual Euclidean coordinates are complex but the distance of the satellite from the center of the Earth should be virtually unchanged. An unexpected change in this distance was used as the error indicator.

3.4. Experimental Procedure

In the first phase of the study, each of the arithmetic operators (+, −, ×, /), the relational operators (<, ≤, =, ≥, >, ≠), and the logical operators (*and*, *or*) were changed into every each other operator in the set. Thus a single + operator generated three mutants with the + operator replaced by −, ×, and / respectively. Some of the resulting mutants failed to compile and were discarded.

In the second phase of the study, operators were selected at random for mutation and the change made was selected at random from the available changes. The total number of mutants generated this way is a very small fraction of the possible mutants for the program. For each of the test programs used in the study the numbers of each type of mutant that survived compilation are shown in Table 1. Table 1 also shows the size of each test program in lines where lines are non-comment source lines of code.

	Program 1	Program 2	Program 3
Lines	34	38	977
Arithmetic	48	45	5 (Fixed)
Relational	25	40	5 (Fixed)
Logical	1	3	0

Table 1 - Numbers Of Mutants and Program Sizes

The error detection performance of data diversity was determined for each mutant. This was done by executing the mutant with test data in an *N*-copy software structure and comparing its error detection performance with that of the original program acting as an oracle. This process was repeated for each available reexpression algorithm. In phase one, the exact reexpression algorithm (rotation about the Earth) was parameterized by the amount of rotation. The resulting *N*-copy system was executed with *N* equal to 32, i.e., the mutant was executed separately with the original data rotated by each of 32 different amounts. Error detection then amounted to checking for any unexpected difference in the 32 different output which should have all been the same.

The general form of the test harness used to evaluate the mutant programs is shown in Figure 3. The purpose of the comparators in the test harness is to determine whether there was any error to detect and to identify false alarms. Although the mutants contained faults, it is not necessarily the case that the fault would manifest itself on each test. Similarly, an error detected by the *N*-copy system did not mean there was an error.

For each mutant, a set of 100 test cases was executed by the *N*-copy system and the oracle, and the error detection performance measured by comparing the number of times the *N*-copy system detected an error to the number of times the oracle detected an error. Finally, this whole process was then repeated 200 times in order to get some information about how the data diverse system's error detection performance varied. During the

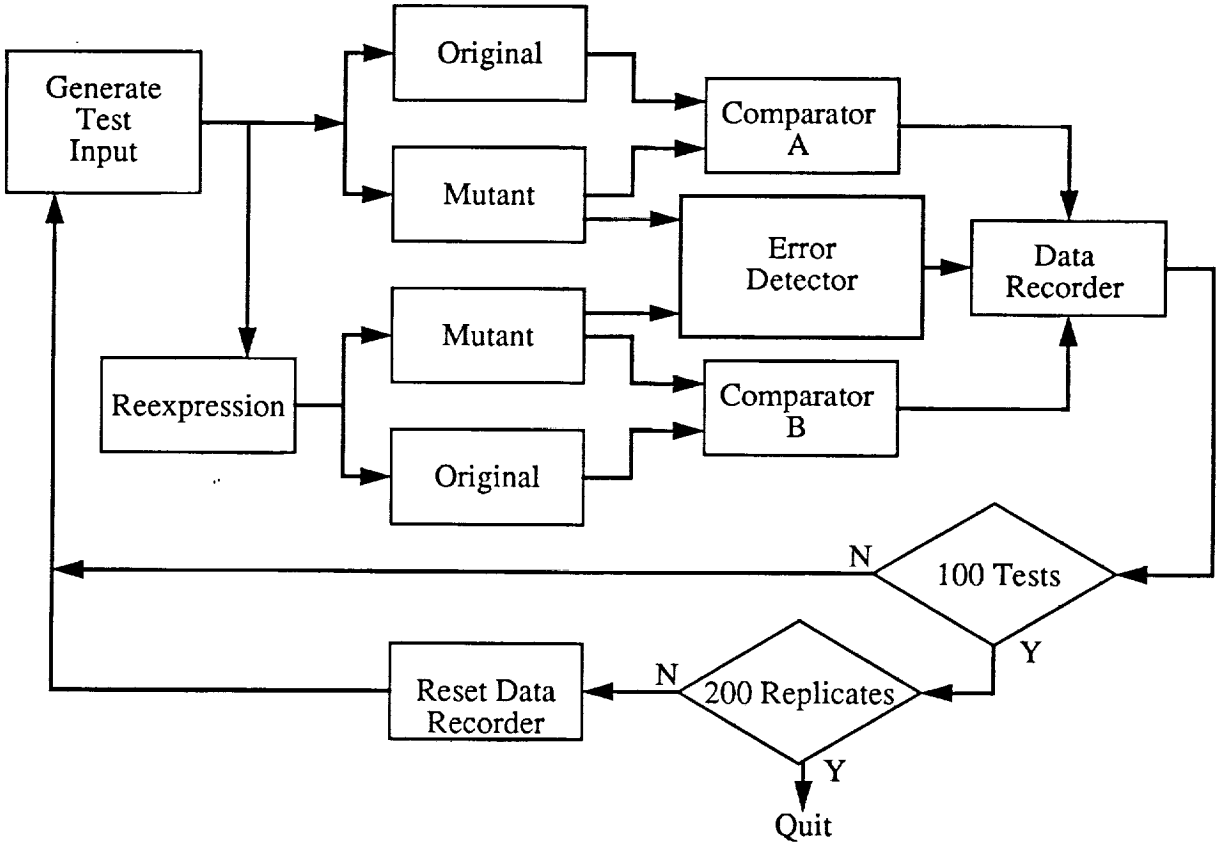

Figure 3 - Evaluation Test Harness

execution of these 200 replicates, the best, worse, and average detection rates for the data-diverse error detector were determined.

4. EXPERIMENTAL RESULTS

4.1. Phase One - Maritime Course Computation

Too many mutants were generated and tested in phase one to permit the results for each mutant to be included in this paper. We have selected from the phase one results

the data for those mutants with the best and the worse error detection performance for each reexpression algorithm. We define best and worst here to be the highest and lowest *average* detection rate, averaged over all 200 replicates. For the first reexpression algorithm, the best and the worst were the same since this reexpression algorithm performed similarly on each replicate. Tables 2 and 3 show the results of testing the arithmetic mutants, and Tables 4 and 5 show the results of testing the relational mutants. In each of these tables, entries expressed as ratios show the number of test cases in which the *N*-copy system detected an error over the number of test cases in which an error occurred. Thus, for example, in Table 2 the maximum data for reexpression algorithm 2 and mutant 11 (center of the table) is 28/40 meaning that of all the replicates that were run, the best detection rate observed for that mutant and reexpression algorithm was 28 detected out of 40 that occurred. Recall that in each replicate 100 tests were run and so the mutant actually failed on 40 out of the 100 tests. A table entry of "NA" means that

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 2 (− → +)	Maximum	40/40	6/40	40/40
	Minimum	40/40	0/40	38/40
	Average	1.000	0.046	0.993
Mutant 10 (× → −)	Maximum	40/40	38/40	40/40
	Minimum	40/40	20/40	37/40
	Average	1.000	0.762	0.995
Mutant 11 (× → +)	Maximum	40/40	28/40	40/40
	Minimum	40/40	11/40	37/40
	Average	1.000	0.472	0.995
Mutant 13 (− → +)	Maximum	40/40	28/40	40/40
	Minimum	40/40	11/40	37/40
	Average	1.000	0.465	0.994
Mutant 15 (− → ×)	Maximum	40/40	14/40	19/40
	Minimum	39/40	1/40	4/40
	Average	0.999	0.183	0.226

Table 2 - Great Circle Computation, Arithmetic Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 1 (− → ×)	Maximum	48/48	13/13	1/10
	Minimum	48/48	6/10	0/17
	Average	1.000	0.876	0.003
Mutant 2 (− → +)	Maximum	48/48	2/7	4/18
	Minimum	48/48	0/13	0/5
	Average	1.000	0.054	0.034
Mutant 5 (+ → ×)	Maximum	100/100	28/99	100/100
	Minimum	100/100	0/17	88/100
	Average	1.000	0.183	0.941
Mutant 19 (− → +)	Maximum	62/62	3/3	1/11
	Minimum	62/62	3/6	0/9
	Average	1.000	0.893	0.002
Mutant 36 (/ → ×)	Maximum	100/100	80/100	100/100
	Minimum	100/100	54/100	97/100
	Average	1.000	0.696	0.997

Table 3 - Rhumb Line Computation, Arithmetic Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 21 (< → >)	Maximum	NA	5/40	6/40
	Minimum	NA	0/40	0/40
	Average	NA	0.047	0.036
Mutant 22 (< → =)	Maximum	NA	4/18	3/17
	Minimum	NA	0/21	0/22
	Average	NA	0.042	0.039
Mutant 24 (< → ≥)	Maximum	NA	5/40	6/40
	Minimum	NA	0/40	0/40
	Average	NA	0.047	0.036
Mutant 25 (< → ≠)	Maximum	NA	4/19	5/23
	Minimum	NA	0/19	0/19
	Average	NA	0.044	0.039

Table 4 - Great Circle Computation, Relational Mutants

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3
Mutant 2 (= → >)	Maximum	20/20	3/3	NA
	Minimum	20/20	0/1	NA
	Average	1.000	0.438	NA
Mutant 4 (= → ≥)	Maximum	40/40	35/40	1/40
	Minimum	35/40	19/40	0/40
	Average	0.973	0.688	0.0004
Mutant 16 (> → <)	Maximum	NA	8/39	4/40
	Minimum	NA	0/40	0/40
	Average	NA	0.085	0.019
Mutant 18 (< → =)	Maximum	NA	5/16	2/13
	Minimum	NA	0/17	0/21
	Average	NA	0.089	0.021
Mutant 39 (< → >)	Maximum	40/40	30/35	1/32
	Minimum	35/40	12/29	0/37
	Average	0.973	0.655	0.001

Table 5 - Rhumb Line Computation, Relational Mutants

for the data produced by that reexpression algorithm, the particular mutant never failed. The mutations used to produce the mutant programs are shown in each table in the leftmost column beneath the mutant number.

Only a single logical operator was present in each of the great circle and rhumb line programs, and the single mutant generated from each did not fail in any of the tests executed.

False alarms, i.e., signaling an error when none was present, occurred very rarely, and the rate was directly determined by the tolerance used in the comparison. Inevitable differences in numeric results sometimes triggered false alarms because the differences were somewhat larger than expected yet still acceptable.

The results of phase one suggest that error detection by data diversity was very effective for arithmetic mutants (Tables 2 and 3). There were considerable differences

between the reexpression algorithms, and, in some cases, differences between the mutants for a given reexpression algorithm. The detection of errors caused by faults in relational operators (Tables 4 and 5) was less effective but the average over all replicates was still nonzero indicating that the fault would eventually be detected, at least during testing. Again there was considerable difference between the reexpression algorithms.

4.2. Phase Two - GPSS Orbit Computation

In phase two, a total of five arithmetic mutants and five relational mutants were generated that survived compilation. There were no logical operators in the program. The results of the testing of the arithmetic mutants is shown in Table 6.

		Reexpression Algorithm 1	Reexpression Algorithm 2	Reexpression Algorithm 3	Reexpression Algorithm 4
Mutant 1 (+ → ×)	Maximum	100/100	0/100	100/100	4/100
	Minimum	98/100	0/100	100/100	0/100
	Average	0.996	0.000	1.000	0.017
Mutant 2 (/ → ×)	Maximum	100/100	0/100	100/100	4/100
	Minimum	99/100	0/100	100/100	0/100
	Average	0.994	0.000	1.000	0.020
Mutant 3 (× → /)	Maximum	100/100	100/100	100/100	32/100
	Minimum	97/100	100/100	100/100	19/100
	Average	0.990	1.000	1.000	0.259
Mutant 4 (× → /)	Maximum	35/35	0/43	38/38	2/46
	Minimum	28/29	0/43	38/38	0/43
	Average	0.986	0.000	1.000	0.0094
Mutant 5 (× → +)	Maximum	100/100	0/100	100/100	3/100
	Minimum	99/100	0/100	100/100	0/100
	Average	0.997	0.000	1.000	0.0140

Table 6 - GPSS Computation, Arithmetic Mutants

Each mutant that was generated by changing a relational operator either failed on every test case that was run or did not fail at all. For those that failed on every test case, none of the failures was detected by the *N*-copy structure. Investigation of this situation revealed that these mutants were generating output that was wildly incorrect and doing so no matter how the data was reexpressed. The test harness was modified to include a simple acceptability check, i.e., that the satellite was actually above the surface of the Earth, and this acceptability test failed on every occasion. Thus although the *N*-copy system was defeated, the software had failed so badly that a trivial additional check detected all of the errors.

The error detection performance observed on the larger program used in phase two was similar to that observed during phase one. The arithmetic mutations caused errors that stood a high chance of being detected but there were considerable differences between reexpression algorithms. The performance on relational mutations, i.e., no detection of programs failing every test case, was surprising and suggests that data diversity might be more useful in detecting errors caused by more obscure faults. Combining data diversity with other error detection techniques, such as the reasonableness check that we used, seems to be a fruitful approach.

5. CONCLUSIONS

We conclude from this study that data diversity has considerable potential for error detection but it is quite variable in its performance. This is confirmation of the same conclusion reached in the pilot study [6]. In the present study, however, we see that the variation occurs across different fault classes and across reexpression algorithms. The fact that for many mutants the *N*-copy systems were able to detect the errors generated with very high probability is encouraging.

The variation in performance can be reduced if more than one reexpression algorithm is used, and by using multiple instantiations of the same reexpression algorithm if it is parameterized. It also appears to be beneficial to combine the method

with other error detection methods. Selection of methods that can be easily automated, such as reasonableness checks, is particularly appropriate. The combined error detection performance of a set of methods would be a profitable area to study.

It is important to note the conclusions we have reached is based on a very small-scale study with many experimental variables deliberately fixed. We cannot extrapolate our results to other application domains or in any other way. General results about the performance of data diversity await further experimentation by ourselves and others.

6. ACKNOWLEDGEMENTS

It is a pleasure to thank Sperry Marine, Inc. of Charlottesville, VA. for permitting us to use samples of their software for this evaluation. John Yancey provided us with substantial technical assistance in understanding the algorithms used. This work was sponsored in part by NASA grant NAG_1-1123-FDP;

REFERENCES

[1] D.J. Martin, "Dissimilar Software In High Integrity Applications In Flight Controls", *Software for Avionics*, AGARD Conference Proceedings, No. 330, pp. 36-1 to 36-9, January 1983.

[2] A. Avizienis, "The *N*-Version Approach To Fault-Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985.

[3] L. Chen and A. Avizienis, "*N*-Version Programming: A Fault-Tolerance Approach To Reliability Of Software Operation", *Digest FTCS-8: Eighth International Symposium on Fault Tolerant Computing*, Tolouse, France, June, 1978, pp. 3-9.

[4] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *Digest FTCS-17: Seventeenth International Symposium on Fault Tolerant Computing*, Pittsburgh, PA, July, 1987, pp. 122-126.

[5] B. Randell, "System Structure For Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[6] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *IEEE Transactions On Computers*, Vol. 37, No. 4, April, 1988.

[7] P.E. Ammann, D.L. Lukes, and J.C. Knight "Applying Data Diversity To Differential Equation Solvers", *submitted to FTCS-21: Twenty First International Symposium on Fault Tolerant Computing*, Montreal, Canada, June, 1991.

[8] P.E. Ammann and J.C. Knight "Data Reexpression Techniques For Fault-Tolerant Systems", Technical Report Number TR90-32, Department of Computer Science, University of Virginia, November, 1990.

[9] S.S. Brilliant, J.C. Knight, and P.E. Ammann, "On The Performance Of Software Testing Using Multiple Versions", *Digest FTCS-20: Twentieth International Symposium on Fault Tolerant Computing*, Newcastle Upon Tyne, UK, June, 1990.

[10] F. Saglietti and W. Ehrenberger, "Software Diversity - Some Considerations About Its Benefits And Its Limitations", *Digest of Papers: SAFECOMP '86, 5th International Workshop on Achieving Safe Real-Time Computer Systems*, France, October, 1986.

[11] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Theoretical And Empirical Studies On Using Program Mutation To Test The Functional Correctness Of Programs", *Proceedings of the Seventh Conference on Principles of Programming Languages*, January, 1980.

APPENDIX B

Applying Data Diversity

To Differential Equation Solvers

Applying Data Diversity To Differential Equation Solvers[†]

Paul E. Ammann
Department of Information Systems and Systems Engineering
George Mason University
Fairfax, VA 22030
(703) 764-4664 FAX: (703) 323-2630
pammann@gmuvax2.gmu.edu

Dahlard L. Lukes
Department of Applied Mathematics
University of Virginia
Charlottesville, VA 22903
dll@virginia.edu

John C. Knight
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
(804) 924-7605 FAX: (804) 982-2214
knight@virginia.edu

ABSTRACT

The control laws for some critical systems are expressed as sets of differential equations. The task of the software is to implement a solution to the control laws. Even though automated aids are heavily used in such implementations, there is no guarantee as to the correctness or reliability of the resulting software. This paper explores a first step in the use of data redundancy to tolerate design faults in differential equation solvers. We distinguish two cases. In the first case, we are interested primarily in the applicability of data redundancy to a wide class of differential equations. In the second case, we are interested in circumstances under which it is reasonable to use an independence model to build highly reliable solvers from moderately reliable components. The paper indicates general tradeoffs and assumptions that underlie this application of the technique and provides data from a pilot study.

Keywords: Data Diversity, Data Redundancy, Design Redundancy, Software Fault Tolerance, Software Reliability.

[†] This work supported in part by NASA grant NAG_1-1123-FDP and NSF grant CCR-9010036.

1. INTRODUCTION

Critical software-based systems continue to be proposed and built. Common examples include medical life support and monitoring equipment, avionics and spacecraft control software, and nuclear power plant shutdown control software. Often these systems implement various control laws expressed as differential equations, and so we must be concerned with the reliability of these implementations.† In this paper we first combine the fault tolerance technique of *data redundancy* [1, 2, 3, 4] (also called *data diversity*) with the rich theory of differential equations to develop general approaches for tolerating design faults in differential equation solvers. We then elaborate additional conditions under which it is justifiable to use independence assumptions to achieve very high reliability levels for such systems.

The most widely studied fault tolerance techniques are based on *design redundancy*, examples of which are *N*-version programming [11] and recovery blocks [27]. In *N*-version programming, a vote on intermediate and/or final results of the redundant implementations is used to determine system output. In a recovery block, an acceptance test on intermediate and/or final values is used to evaluate the results of successive alternate implementations. In addition to being used for fault tolerance, the design redundant structures are also used via *comparison checking* or *back-to-back testing* [7, 10, 28, 30] for software fault removal. Design redundant systems suffer from a variety of difficulties [8, 9, 13, 17, 18]; see [15] for a thorough review. The most serious of these problems is modeling the effect of component failure on the failure characteristics of the entire system; the basic difficulty is that the models from the analogous hardware systems simply do not apply [13]. We return to the issue of modeling in section 2.

An alternate approach to the detection and tolerance of software faults, and the approach that is the topic of this paper, is *data redundancy*. The difference between design and data redundancy can be summarized as follows: design redundancy employs *related* designs on the *same* input; data redundancy employs the *same* design on *related* inputs. Data redundant systems are economically attractive and potentially very effective [2].

The basic framework for data redundancy is as follows: In a data redundant system, only

† Concern about correct operation is not limited to reliability; see, for example, [19].

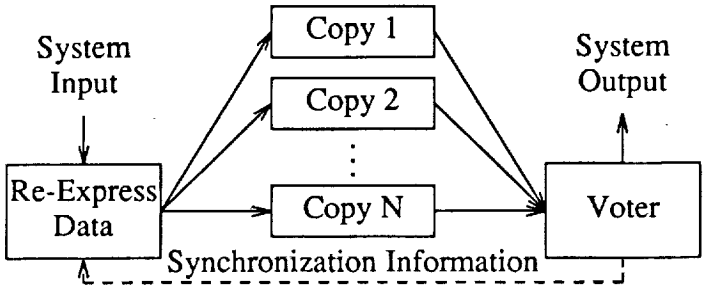

Fig. 1: N-Copy Programming.

one implementation of a program is developed from the specification.† A given input, x, is used to derive N alternate inputs, $x_1, x_2, \ldots, x_N$, that satisfy some application dependent equivalence property. The program, P, is run on the alternate inputs, x_i, to produce $P(x_1), P(x_2), \ldots, P(x_N)$. A decision algorithm is employed to evaluate the (possibly different) outputs, $P(x_i)$. The manner in which alternate inputs are derived is the fundamental determinant of the success of data redundancy.

Data redundant program structures parallel the design redundant structures. *N-Copy Programming* is based on N-Version programming, except that the vote is used to select from the outputs produced by the execution of the same implementation on a set of alternate inputs. The *retry block* is based on the recovery block, except that when the acceptance test fails, the same implementation is retried with an alternate input. Fig. 1 illustrates the semantics of N-Copy programming; fig. 2 illustrates the semantics of the retry block.

In this paper, we discuss the application of data redundancy to differential equation solvers. In section 2, we address the modeling problem: we outline the requirements on data redundant systems such that the one is able to meaningfully predict the reliability of a system based on the reliability of the system's components. In section 3, we discuss, first by way of example and then with a general theory, the way in which data redundancy can be applied to differential equation

† We adopt the following simple but general model to focus the presentation: During testing or during operation, the program in question, denoted P, executes on a specific input, x, to produce an output, $P(x)$.

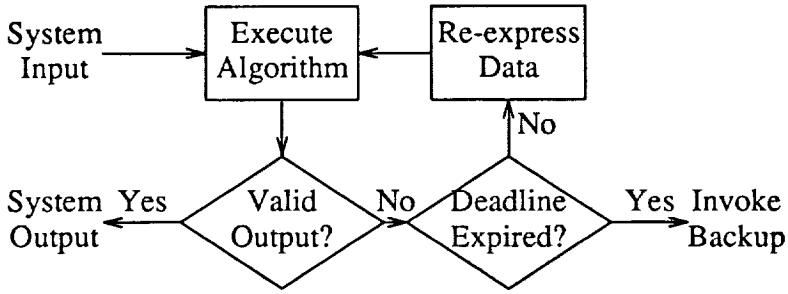

Fig. 2: Retry Block.

solvers. In section 4, we join sections 2 and 3 by recasting the example of section 3 so that the reliability of the solver can be probabilistically guaranteed. In section 5, we give results from a pilot study that illustrates the behavior of the example. Section 6 contains conclusions.

2. THE MODELING PROBLEM

In hardware systems, it is well understood how to construct systems of high reliability from components of moderate reliability [29]. For example, in *N*-Modular Redundant (NMR) hardware systems, *N* replicates of a hardware module are supplied so that the failures of some subset can be tolerated. Similarly, in standby sparing systems, replicates of a component can take over in the event that the primary component fails. The key factor for the success of structures such as NMR and standby sparing is that it is reasonable to model component failures as independent events; the probability that two components fail simultaneously is assumed to be the product of the individual failure probabilities. It is indeed fortunate that hardware models can take advantage of the independence assumption; if the assumption did not apply, actual hardware reliability would be substantially lower.

N-version programming parallels the hardware NMR structure. Similarly, the recovery block parallels the hardware standby sparing model. Unfortunately, as has been shown empirically [18], the independence model is not appropriate for *N*-version programming or the recovery block. The difficulty is that the probability of failure for a randomly chosen software component is a function

of the input being processed; some inputs are simply more likely to trigger design faults than others. Models which take account of this dependence, such as the multi-version model developed by Eckhardt and Lee [13], generally (although not always) predict substantially lower reliabilities. Worse, such models yield average results that are parameterized by an unknown failure intensity distribution. In general, these models are not useful for predicting the properties of a specific system.

In [2] it was shown that the Eckhardt and Lee model could be adapted to describe the performance of data redundancy in general applications. Thus, in the absence of additional information, the independence model is *not* appropriate for general data redundancy applications. In [2], it was further shown that the program structure employed, either a retry block or an N-Copy system, was much less important to the performance of data redundancy than the ability of the re-expression algorithm to exit the failure region.

However, we now turn to a basic point of this paper: it is possible, for certain applications, to employ data redundancy in such a way that it *is* reasonable to apply an independence model. In such cases, it is possible to take software components of modest reliability and construct a system of substantially higher reliability. Below we enumerate methods by which one can justify the use of an independence model. We highlight the assumptions underlying these methods. It is important to recognize that we are not supplying a panacea for the problem of software reliability; our claim is only that, with care, the independence model can reasonably be applied to tolerate design fault in an important set of applications. This paper considers the independence model in the differential equations domain; other researchers have applied random algorithms in various other areas [5, 20].

2.1. Data Redundancy And The Independence Model

How is the independence model made applicable? Suppose that we are given a usage distribution over the domain of interest.[‡] Through life testing [23, 26], it is possible to bound the failure probability due to a design fault on a randomly chosen input. That is, if an input is randomly selected from the domain according to the usage distribution, then, with confidence C, we can be

[‡] For the following analysis, it is not necessary that the postulated usage distribution precisely match the actual usage distribution. However, as will become apparent in Section 5, selecting inputs from a significantly different distribution does affect performance.

sure that the probability of failure does not exceed some fixed value, say p. Random selection yields the independence property; the probability that two randomly selected inputs will simultaneously cause failure is p^2.

It may not appear that we have made much progress. Although it is true that by randomly selecting inputs we can be sure that the different invocations of the software are failing independently, we must be able to use the outputs produced by the randomly selected inputs. After all, if we are given an input x, we are required to produce the program output $P(x)$, not $P(y)$, for some random y. However, in certain cases, it turns out that we can use additional information to relate a random input, y, back to the original input, x. Below we outline two methods for achieving this goal.† Both techniques are applicable to the differential equations arena.

Single Input Method

One basic method is to select random parameters that map x to a single arbitrary y in the domain, and then use those same parameters to map $P(y)$ back to $P(x)$. If x is a vector instead of a scalar, it is important that the parameters represent a sufficiently rich transformation. For example, if x is a point in R^3, then the transformation needs to able to map x to some three dimensional subregion of R^3. Clearly, a two parameter transformation is insufficient to guarantee the equivalent of random selection in R^3.

Multiple Input Method

Another basic method is to choose a random input y_0 independently of the given input x, and then choose a set of other inputs, $y_1, y_2, ..., y_N$. The criterion for selection of the y_i is that the multiple outputs $P(y_i)$ can be combined in such a way that, in the absence of failure, there is a direct relation to the output $P(x)$. *I.e.*, we need a function $f(\,)$, such that, in the absence of failures, $f(P(y_0), P(y_1), ..., P(y_N)) = P(x)$. This powerful technique has been explored by Lipton [20] for polynomials and by Ammann and Knight [3, 4] for simple trigonometric functions. We review these applications below.

† We note that [5] exploits this idea in a slightly different manner in the design of program checkers.

Identity Function

The first example we give for the single input method is very simple, and is of explanatory value only. The example appears in [20]. Suppose that we wish to reliably compute the identity function, $f(x) \equiv x$, and that all that we have available is a (possibly) faulty implementation, P, with the property that $P(x)$ actually differs from x with probability p on randomly chosen x. If we compute $P(x+r)$ for randomly chosen r, we can be sure that the probability that P will fail on both input x and input $x+r$ is p^2. Note that, with probability $1-p$, we can recover $f(x)$ from $P(x+r)$ by subtracting r.

Polynomials

We now address a more substantive example. A comprehensive treatment of this example may be found in [20]. Suppose that we wish to multiply pairs of numbers more reliably than we are assured for a given implementation of multiplication. Such an application might arise, for example, in cryptography, where it is necessary to perform modulo arithmetic on numbers with hundreds of digits. Consider the computation $A \times B$. Select two random values, X_1 and Y_1 and compute $X_2 = A - X_1$ and $Y_2 = A - Y_1$. We may rewrite the product $A \times B$ as $X_1 \times Y_1 + X_1 \times Y_2 + X_2 \times Y_1 + X_2 \times Y_2$. Note that we have rewritten the product of A and B as the sum of four other products. Due to independent selection, we may be sure that failure on each of the four resultant multiplications is independent of failure on the multiplication of A and B. In his analysis in [20], Lipton shows that the independence property can be maintained when the transition is made from algebra to typical machine representations. In addition, Lipton extends the example to demonstrate reliable computation of polynomials.

Sine Function

Our final example is repeated from [3]. It is a data redundant computation of the sine function. Assume we have an implementation of the sine function that is known to work over a large percentage of it inputs; the failure probability for the sine function on a randomly chosen input, x, is p, where $p << 1$. To compute $\sin(x)$, we use the two trigonometric identities

$$\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$$

$$\cos(a) = \sin(\pi/2 - a)$$

to rewrite

$$\sin(x) = \sin(a)\sin(\pi/2-b) + \sin(\pi/2-a)\sin(b)$$

in which a and b are any two real numbers such that $a+b = x$. Suppose that $\sin(x)$ is computed using three independent decompositions for x obtained by using three different values each for a and b, and that a simple majority voter selects the output. Using the worst case assumption that all incorrect answers appear identical to the voter, a conservative estimate for the probability of computing an incorrect value for $\sin(x)$ is on the order of $48p^2$.

Series Expansions

If we are content to describe a function with a series expansion over either polynomials or sines and cosines, we can use the techniques above to calculate arbitrary functions with very high reliability given components of moderate reliability and assumptions about the correctness of the basic arithmetic operations.

3. APPLYING DATA REDUNDANCY TO DIFFERENTIAL EQUATIONS

In this section, we wish to elaborate the general ways in which the underlying theory of differential equations may be exploited in data redundant implementations of differential equation solvers. We begin with a simple example of a harmonic oscillator, and then proceed to more general theory based on Lie symmetries. In this section, we concentrate exclusively on the differential equations; in the next section we tie the results reported below to the independence property described in the previous section.

3.1. An Example

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad , \quad x(0) = x_0 = \begin{bmatrix} x_1{}^0 \\ x_2{}^0 \end{bmatrix} .$$

We wish to compute $x(T)$.

We wish to exploit the symmetry of the given o.d.e. by letting the group $G = R^2 \times R^+$ act on the data to be fed into the o.d.e. solver used to provide the computed value. The ultimate objective is to improve the reliability of the accepted computed value.

The group G will be tied into the ode solver which when passed parameter values $[\alpha,\beta,\gamma] \in G$ will be referred to as the $[\alpha,\beta,\gamma]$-solver. This will be done in such a fashion that, corresponding to the group identity $[0,0,1] \in G$, the $[0,0,1]$-solver is the solver for the above given harmonic oscillator. The main objective is to produce multiple evaluations for $x(T)$, each obtained by a (small) finite number of calls to the $[0,0,1]$-solver and any $[\alpha,\beta,\gamma]$-solver on appropriate arguments.

Consider the initial-value problem

$$\frac{d}{d\sigma}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \gamma\begin{bmatrix} \alpha & \beta+1 \\ -\beta-1 & \alpha \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad , \quad y(0) = y^0 = \begin{bmatrix} y_1{}^0 \\ y_2{}^0 \end{bmatrix} .$$

Denote its solution value at σ as $P[\alpha,\beta,\gamma](\sigma,y^0)$ which we identify with the value returned by an exact $[\alpha,\beta,\gamma]$-solver.

Proposition For each $[\alpha,\beta,\gamma] \in R^2 \times R^+$ and $\sigma \in R$,

$$P[0,0,1](\gamma\sigma,P[\alpha,\beta,\gamma](-\sigma,P[0,0,1](T,P[0,0,1](-\gamma\sigma,P[\alpha,\beta,\gamma](\sigma,x_0))))) = x(T),$$

for every α,β,γ and σ. If all return the approximate same value of $x(T)$ then we have increased confidence that $P[0,0,1]$ is working for $t_0 = 0$, x_0 on $0 \le t \le T$.

It is easy to show that the $[\alpha,\beta,\gamma]$-solver can move any initial-data point in the (t, x)–*space* to any other as long as x is not zero.

3.2. General Theory

Assume that the equations have been cast into the first order system form of the the initial-value problem

$$\dot{x} = f(t, x) \quad , \quad x(t_0) = x_0$$

where x is a n-vector state variable in R^n and, for simplicity, assume that f is a smooth function of its argument in $R \times R^n$. The classical theory of differential equations ensures the existence of a unique solution $x = \eta^f(t, t_0, x_0)$ satisfying the differential equation and the initial condition that $\eta^f(t_0, t_0, x_0) = x_0$. (The domain and properties of this general solution η^f are discussed in [21].) For each choice of input data (t_0, x_0) a good differential equation solver implemented correctly provides a discrete approximation to the resultant trajectory $(t, x(t))$ in $R \times R^n$ with

$x(t) = \eta^f(t, t_0, x_0)$.

The idea of data redundancy discussed earlier in this paper leads to questions of the Lie theory type concerning symmetries of differential equations. This is best explained in terms of the diagram of fig.3. (Also see fig.2.)

Extending the notation - scheme of Example 3.1, we call the implementation of a numerical integration algorithm applied to a differential equation given by f a $[f\,]$–*solver.* Hence, if all is working well, if the $[f]$-solver is given input data (t_0, x_0) it produces the numerical approximation to the trajectory $(t\,, \eta(t, t_0, x_0))$ on the interval $t_0 \le t \le t_1$ and in particular computes $(t_1\,, x_1)$ where $x_1 = \eta^f(t_1, t_0, x_0)$.

The basic data re-expression question raised is: Can this numerical solution be obtained by feeding the $[f]$-solver some other input data $(\tilde{t}_0, \tilde{x}_0)$? Motivation for the question might arise from some doubt about the validity of the result returned by the $[f]$-solver in response to the original input (t_0, x_0). Once obtained, $(\tilde{t}_0, \tilde{x}_0)$ would be used to run the $[f]$-solver a second time to get $(\tilde{t}_1, \tilde{x}_1)$ and the results would then be transformed back to provide the solution to the original problem. As did Sophus Lie, who used another ordinary differential equation to transform the original problem but for the purpose of finding a change of coordinates that would reduce the dimension of the original equation as a step in finding the general solution to the original equation (see

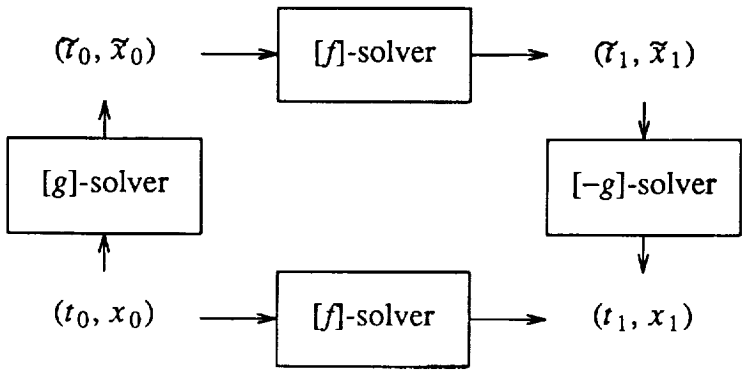

Fig. 3: Data Re-Expression Via Lie Symmetries.

[6, 25]), we choose to transform the initial data by means of another differential equation. The equation $\frac{dz}{d\sigma} = g(z)$, this time is autonomous, i.e., independent of σ with z a variable in R^{n+1}. The invariance requirement, which is that with $(\tau_0, \xi_0) = \eta^g(\sigma, 0, t_0, x_0)$, $(t, \eta^f(t, \tau_0, \xi_0))$ transform back via $\eta^g(-\sigma, 0, \cdot, \cdot)$ to give the solution $(t, \eta^f(t, t_0, x_0))$ to the original problem then is equivalent to the statement that the diagram in fig. 3 be commutative. Due to space limitations we omit details but it can be shown that this commutativity is equivalent to the imposition of the partial differential equation

$$[\hat{f}, g] = 0$$

where $\hat{f} = f \otimes 1$ and the bracket $[\ ,\ ]$ is the Jacobi bracket given by the formula

$$[h_1, h_2] = (Jh_2)h_1 - (Jh_1)h_2$$

in which $(Jh)_{ij} = \frac{\partial h^i}{\partial x_j}$ is the Jacobian matrix of any vector field h. Here, the superscript i on h denotes the *ith* coordinate of h. A solution g is called an *infinitesimal generator of a one-parameter family of symmetries* of f, or just an *infinitesimal generator* or just *generator* for f.

Having cast the basic re-expression question into the language of Lie theory we can now make some general statements: The infinitesimal generators of f constitute a Lie algebra relative to the Jacobi bracket product, generally infinite dimensional. The infinite dimensionality is good news for it indicates that the potential for data re-expression is extensive. (Since g is autonomous the inverse transform is accomplished by application of the $[-g]$-solver).

Of course $[h, h] = 0$ for every vector field h, hence $\hat{f}$ is always an infinitesimal generator for f. Therefore $g = \hat{f}$ provides trivial re-expression, namely, the restarting of the $[f]$-solver using the most recently computed value of (t, x) computed as initial input. In particular it provides the simple test wherein one restarts the $[f]$-solver with initial data (t_1, x_1) where x_1 is the value returned for $\eta^f(t_1, t_0, x_0)$ and compares it with the value ξ_0 returned for $\eta^f(t_0, t_1, x_1)$. If $|\xi_0 - x_0|$ is not small the approximate solution for $\eta^f(t, t_0, x_0)$ produced by the $[f]$-solver is rendered suspect. (We assume that the solver is written so that it computes backward as well as forward trajectories.)

Fortunately the entire Lie algebra of symmetries of f need not be computed. (It can be constructed from η^f but this construction is only of theoretical interest.) Once a single generator g for f is found its $[g]$-solver with input data (t_0, x_0) produces a one parameter family,

$(\tilde{t}_0, \tilde{x}_0)_\sigma = \eta^g(\sigma, 0, t_0, x_0)$, of re-expressed values of the initial data. If two or more generators are found in addition to $\hat{f}$ then others can easily be obtained via bracket products and linear combinations since they generate a subalgebra. Of course the more generators one finds the greater is the capability for doing the re-expression. Even though it is possible to transform the initial data to arbitrary values in specific instances, in general, this is not the case.

For special differential equations, namely those arising from a single scalar higher order differential equation methods are available for computing infinitesimal generators and the Lie algebras obtained are finite dimensional. (see [6, 25].)

In applications a finite and perhaps small number of generators might prove adequate. One might restrict the choice to a parameterized family as illustrated by Example 3.1 of this article where

$$f(t, x_1, x_2) = \begin{bmatrix} x_2 \\ -x_1 \end{bmatrix} \quad , \quad g_{\alpha\beta\gamma}(s, x_1, x_2) = \begin{bmatrix} \gamma \\ \alpha x_1 + (\beta + 1)x_2 \\ -(\beta + 1)x_1 + \alpha x_2 \end{bmatrix}$$

Although it is a simple matter to write down the exact solutions to the differential equations given by these vector fields, since they are linear, such generally is not the case. We include the following theorem which gives an algebraic criterion for commutativity within a class of nonlinear differential equations whose members are not readily solvable except by numerical methods.

Theorem

Let A and $\tilde{A}$ be real $n \times n$ matrices having the same eigenvalues, i.e., they have the same characteristic polynomial. Select any vectors λ , ζ , $\tilde{\zeta}$ in R^n.

Then the vector fields

$$f(x) = e^{\lambda^T x A}\, \zeta \qquad , \qquad \tilde{f}(x) = e^{\lambda^T x \tilde{A}}\, \tilde{\zeta}$$

commute if

$$A^k \zeta \lambda^T [\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}] = \tilde{A}^k \tilde{\zeta} \lambda^T [\zeta, A\zeta, \cdots, A^{n-1}\zeta] ,$$

$(k = 1, 2, \cdots, n-1)$.

Proof. In [21] it is shown that for any $n \times n$ matrix A, and ζ in R^n,

$$e^{tA}\zeta = [\zeta, A\zeta, \cdots, A^{n-1}\zeta][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}e^{tB}y^0$$

for some $n\times n$ real matrix B and y^0 in R^n depending on only the characteristic polynomial of A, i.e., on the eigenvalues of A with their multiplicities. Consequently, in the notation of the theorem being proved,

$$f(x) = [\zeta, A\zeta, \cdots, A^{n-1}\zeta][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}e^{\lambda^T xB}y^0$$

$$\tilde{f}(x) = [\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\zeta][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}e^{\lambda^T xB}y^0$$

for some y^0 in R^n and $n\times n$ matrix B. Letting $f'(x)$ denote the Jacobian matrix of $f(x)$ we compute

$$f'(x)\tilde{f}(x) = [\zeta, A\zeta, \cdots, A^{n-1}\zeta][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}Be^{\lambda^T xB}y^0\lambda^T$$
$$[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}e^{\lambda^T xB}y^0.$$

Therefore for $f'(x)\tilde{f}(x) = \tilde{f}'(x)f(x)$ it is sufficient that

$$[\zeta, A\zeta, \cdots, A^{n-1}\zeta][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}Be^{tB}y^0\lambda^T[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}] =$$
$$[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}][y^0, By^0, \cdots, B^{n-1}y^0]^{-1}Be^{tB}y^0\lambda^T[\zeta, A\zeta, \cdots, A^{n-1}\zeta]$$

This equation can be rewritten as

$$\frac{d}{dt}e^{tA}\zeta\lambda^T[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}] = \frac{d}{dt}e^{t\tilde{A}}\tilde{\zeta}\lambda^T[\zeta, A\zeta, \cdots, A^{n-1}\zeta]$$

which is equivalent to

$$e^{tA}A\zeta\lambda^T[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}] = e^{t\tilde{A}}\tilde{A}\tilde{\zeta}\lambda^T[\zeta, A\zeta, \cdots, A^{n-1}\zeta].$$

But writing out the exponential functions in series shows that the above equation is equivalent to the matrix equations

$$A^kA\zeta\lambda^T[\tilde{\zeta}, \tilde{A}\tilde{\zeta}, \cdots, \tilde{A}^{n-1}\tilde{\zeta}] = \tilde{A}^k\tilde{A}\tilde{\zeta}\lambda^T[\zeta, A\zeta, \cdots, A^{n-1}\zeta]$$

$k = 0, 1, 2, \cdots$). Since A and $\tilde{A}$ are assumed to have the same characteristic polynomial we we see by the Cayley-Hamilton theorem that if the equations hold for $k+1 = 1, 2, \cdots, n-1$ then they will automatically hold for the remaining values of k. This concludes the proof.

In concluding this section we mention some related work published by one of the authors (Lukes [22]) which takes an arbitrary linear controllable differential equation and asks: What linear input, output and feedback filters can one wrap around the dynamical system in a manner that leaves the overall input-output characteristics invariant but alters the inputs that the original dynamical system would experience? There it was shown that each controllable system admits its own Lie group of such filters. Moreover the group can be computed. This group expresses the

original system's potential for fault avoidance. The present article continues a variation of that theme which started in [2] where it was discussed in the context of general programs. The hypothesis being proposed suggests that there is a potential for designing software and perhaps some hardware in a manner where the parameters enter in such a fashion that at least certain classes of faults could be reasonably tolerated using ideas discussed in other sections of this paper.

4. RELIABILITY GUARANTEES

We now rework the example given in the previous section to show how one can go about developing a system that offers probabilistic guarantees of reliability. We show that it is possible to employ data redundancy in such a way so as to justify applying the independence model discussed earlier. It should be noted that many of the re-expression techniques cited below are standard practice in the numerical methods area [14]. The focus here is to use these techniques in such a way so as to allow the use of the independence model described in Section 2.

4.1. The Oscillator Revisited

The differential equation describing the oscillator accepts the following transformation:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \alpha & \beta + 1 \\ -\beta - 1 & \alpha \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

I.e., the transform in α and β maps solutions of the differential equation to other solutions of the differential equation. Suppose we are given input x. We can choose random values for α and β and derive a new input y. After we compute $P(y)$, we can, in the absence of failure, use the inverse transformation to retrieve the desired quantity $P(x)$. It is important to note that the distribution for y is *not* the usage distribution. As will be shown in the pilot study in Section 5, this seemingly unimportant detail destroys the ability of the model from Section 2 to accurately predict the failure probability of a data redundant system. However we can avoid the problem by selecting y from the usage distribution and then deriving the quantities α and β that map x to y. With this mode of operation, the independence model does apply.

A standard technique that applies to linear equations is superposition. We may exploit superposition for fault tolerance in such a way that the independence model applies. However, some care is required. Consider an input x. Suppose that we break x into two inputs y and z

where $x = y+z$. If we choose y from the usage distribution and then compute $z = x-y$, we are justified in using the independence model. If we instead compute y uniformly from the interval $[0, x]$, the independence assumption no longer applies. Again, the difference between these two approaches is illustrated in the pilot study reported in Section 5.

As a final note, we have only addressed the problem of transforming x. We note that we can handle the time parameter t in a number of standard ways. The most straight forward is to note that the we may choose another time interval t_1 at random, and then derive $t_2 = t-t_1$. We can then compute $P(P(x,t_1),t_2) = P(x,t)$ Another standard technique for dealing with faults that manifest themselves in specific values of the t parameter is to adjust the step size Δt.

4.2. General Theory

The theory in Section 3 explains how to derive various differential equations to transform the initial input. In general, the reliability of the solver implementing these alternate differential equations may be suspect. However, in some cases these differential equations are sufficiently simple that they correspond to simple algebraic manipulations. For differential equations of sufficient interest, *i.e.* differential equations in critical control systems, it is not unreasonable to expect that the Lie symmetries be known. And since any Lie group may be reparameterized, it may be reasonable to rework the equations so that "standard" transforms, such as scaling, rotation, and translation are admitted. The complete details of such an approach constitute an important open question.

5. PILOT STUDY

In this section we describe a pilot study that investigates the behavior of various data redundancy strategies applied to a differential equation solver.

5.1. Design

We consider an implementation of a specific differential equation solver, namely a Runge-Kutta 4th Order solver [14]. We note that the selection of any other standard solver would have been appropriate. We investigate the behavior of the solver on the harmonic oscillator example presented in Section 3.1.

The parameter we vary over the course of the experiment is the data re-expression algorithm. We explore 5 different re-expression algorithms; 2 are based on the independence model discussed in section 2, and the other 3 are not. By including both types of strategies, the study illustrates the general properties of independence model discussed previously.

The first strategy, labeled '1' in subsequent figures, was outlined in the example in Section 3.1. The program P, along with three parameters α, β and γ, is used in the re-expression algorithm. The second two strategies are based on the observation that the two parameter transform described in Section 4.1

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \alpha & \beta+1 \\ -\beta-1 & \alpha \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

maps solutions of the oscillator onto other solutions. In the strategy labeled '2', the parameters α and β are chosen at random; the alternate input y is then derived from the given transformation. In the strategy labeled '3', the alternate input y is chosen at random, and α and β are derived according to the transformation. We will see that this seemingly slight difference between the two methods can have a dramatic impact on performance. The fourth and fifth strategies are based on the fact that for linear systems, sums and scalar multiples of solutions are also solutions. We derive two inputs from the original input; run the solver on both, and then sum the two results. In the strategy labeled '4', the alternate inputs are derived by choosing two scale factors, α and β at random. In the strategy labeled '5', one alternate input is chosen at random, and the other is derived by subtracting it from the original input. Again, we will see that this seemingly slight difference between the two methods can have a dramatic impact on performance.

For this experiment, the focus is on examining the properties of various data re-expression algorithms with respect to the independence model. For that reason, we have chosen *not* to apply the model to a large or representative variety of software faults. If we are able to select inputs apparently at random according to the usage distribution, then the characteristics of the particular software faults are irrelevant to the performance of the data redundant system. If the inputs are not randomly selected, the details of the various faults are then vitally important. The goal of the experiment here is to highlight this distinction, not fully evaluate the consequences of failing to meet requirements of the independence model.

For the reasons outline above, the solver was seeded with a single generic fault that caused failure if the tolerance between the two components of the solution dropped below a certain threshold. Such a fault could arise in a variety of locations in typical implementation; for example, such a fault could arise in a relative error test for convergence. In addition, the tolerance level provided a convenient handle to adjust the failure rate of the fault to any desired level. In short, the generic fault is ideal for the experiment described here. For experiments designed to evaluate general data re-expression algorithms on arbitrary faults, a method such as mutation analysis [12] or fault-based testing [24] would be appropriate.†

5.2. Results

We now present the results of evaluating the 5 data redundant strategies described above. The particular mechanism that we use for analysis is the retry block with a single retry. In fig. 4,

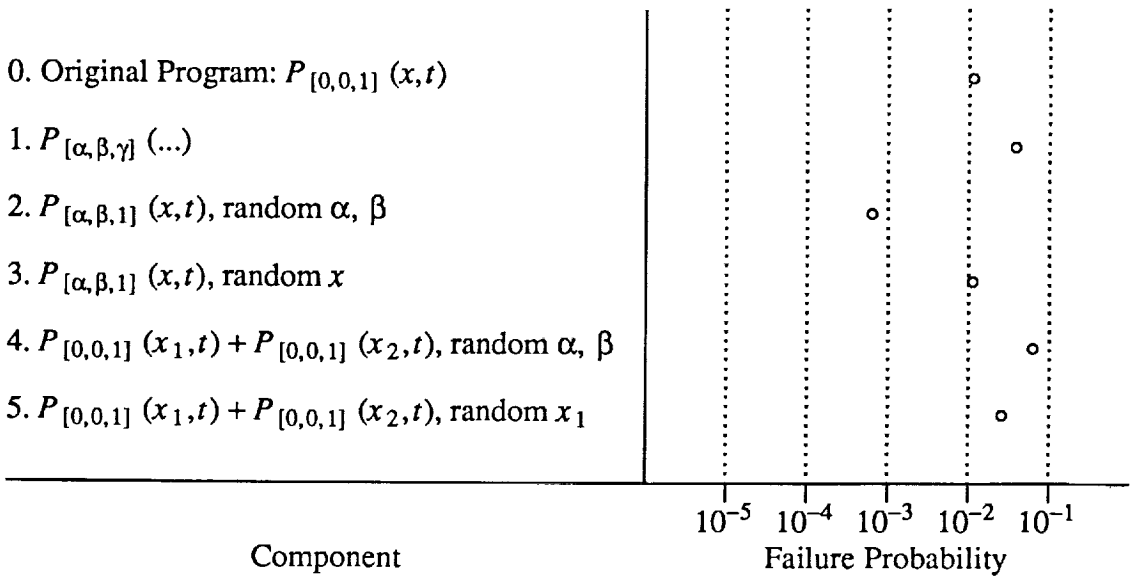

Fig. 4. Observed Component Failure Probabilities

† See, for example, [16].

we present the observed component failure probabilities for each of the five strategies. Confidence bounds are not reported on the figure because the bullets that represent each data point encompass the 90% confidence bound. The strategy labeled '0' corresponds to the differential equation solver operating on the original input without any application of data redundancy. Several points in fig. 4 warrant notice. First, for strategies '1', '4', and '5', the failure probability of the component exceeds that of the original program. This is because multiple executions of P are required for each of these strategies. Second, for strategy '3', the failure probability of the component matches that of the original program. This is reasonable, since only a single execution of P is required, and that execution occurs on an input chosen according to the usage distribution. Finally, the failure probability for the component in strategy '2' is substantially lower than for the original program. How is this possible? It turns out that the data re-expression algorithm for strategy '2' happens to map to inputs in a region of the domain with a lower failure probability for the given fault. This is entirely a function of the particular fault; for other faults the situation could easily be reversed. Thus the relatively low value of the failure probability for strategy '2' should not be viewed as a desirable result.

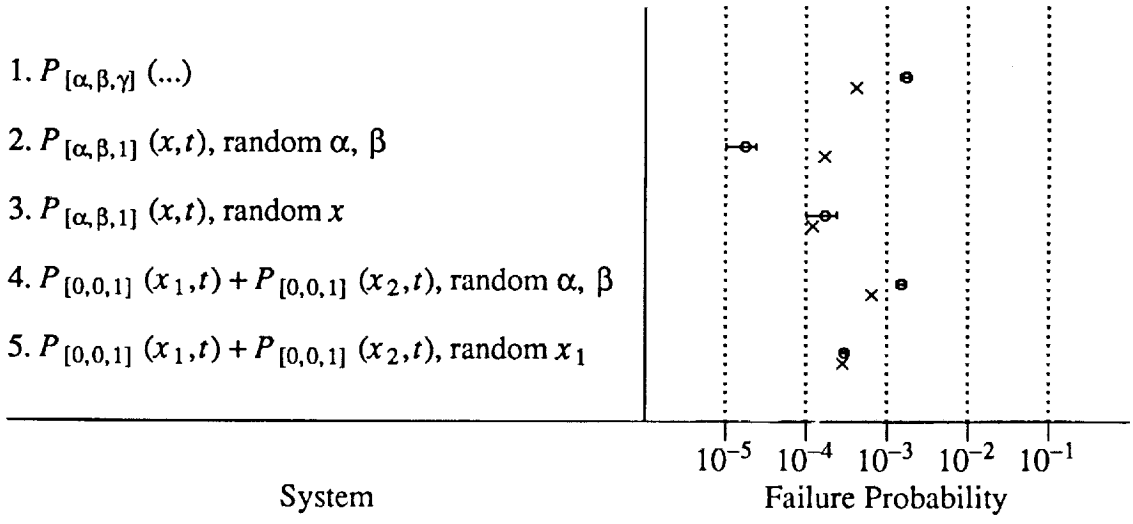

Fig. 5. Single Retry System Failure Probabilities: Observed (O) and Independence Model (×)

In fig. 5, we present failure probabilities for systems built from the components enumerated in fig. 4. Each system is a retry block in which component '0' executes first. If component '0' fails, then a data redundant component executes. Entries in fig. 5 are the probabilities that both of these computations fail coincidentally; *e.g.* we define this event as system failure. Entries marked by a circle are the observed system failure probabilities. A 90% confidence bound is placed around each observation. Entries marked by a cross are the failure probabilities predicted by an independence model. These values are simply the product of the corresponding entries from fig. 4.

There are several important points to note about fig. 5. First, the failure properties of the two strategies ('3' and '5') that explicitly selected alternate inputs according to the usage distribution agree with the independence model. Second, for the strategies in which we selected random parameters instead of random inputs ('2' and '4'), note that the independence model clearly does *not* apply. In the case of strategy '4', the performance falls short of the independence model. In the case of strategy '2', the performance in much better; again the reader is cautioned that this is purely an artifact of the fault under study. Other faults would produce other behavior.

6. CONCLUSIONS

We have outlined a general theory for applying data redundancy to differential equation solvers. On cost considerations alone, these should be considered serious candidates for fault-tolerance. They are also very strong candidates for error detection at test time.

We have given a simple example of such an application. The example demonstrates that the overhead to apply such techniques is not prohibitive, and that many degrees of freedom are possible in specifying alternate data. We have evaluated the example experimentally under a variety of data re-expression algorithms. The pilot study illustrates the areas of concern in implementing a data redundant system under an independent failures model.

Design redundant systems cannot guarantee high system reliability based on moderate component reliability. In general, data redundant systems suffer from the same limitation. However, specific applications of data redundancy can employ random algorithms to achieve guarantees of very high system reliability. We have taken a first step in applying random algorithms to the differential equations area.

REFERENCES

[1] P.E. Ammann, "Data Redundancy for the Detection and Tolerance of Software Faults", *Proceedings Interface '90*, East Lansing, MI, May, 1990.

[2] P.E. Ammann, "Data Diversity: An Approach To Software Fault Tolerance", PhD Dissertation, University of Virginia, January, 1988.

[3] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *Digest FTCS-17: Seventeenth International Symposium on Fault Tolerant Computing*, Pittsburgh, PA, July, 1987, pp. 122-126.

[4] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach To Software Fault Tolerance", *IEEE Transactions On Computers*, Vol. 37, No. 4, April, 1988.

[5] M. Blumen and S. Kannan, "Program Correctness Checking ... and the Design of Programs That Check Their Work", *Technical Report TR-88-013*, International Computer Science Institute, Berkeley, CA, December, 1988.

[6] G.W. Blumen and S. Kumei, *Symmetries and Differential Equations*, Springer-Verlag, New York: 1989.

[7] S.S. Brilliant, "Testing Software Using Multiple Versions", PhD Dissertation, University of Virginia, January, 1988.

[8] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software", *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, November, 1989.

[9] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis of Faults in an *N*-Version Software Experiment", *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, February, 1990.

[10] S.S. Brilliant, J.C. Knight, and P.E. Ammann, "On The Performance of Software Testing Using Multiple Versions", *Digest FTCS-20: Twentieth International Symposium on Fault Tolerant Computing*, Newcastle Upon Tyne, UK, June, 1990.

[11] L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation", *Digest FTCS-8: Eighth International Symposium on Fault Tolerant Computing*, Tolouse, France, June, 1978, pp. 3-9.

[12] R. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help For the Practicing Programmer", *IEEE Computer*, April, 1978.

[13] D.E. Eckhardt and L.D. Lee "A Theoretical Basis For The Analysis Of Multiversion Software Subject To Coincident Errors", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985.

[14] R.W. Hornbeck, *Numerical Methods*, Quantum Publishers, New York: 1975.

[15] J.C. Knight and P.E. Ammann, "Design Fault Tolerance", to appear in *Reliability Engineering and System Safety*, 1990.

[16] J.C. Knight and P.E. Ammann, "On The Effectiveness of Data Diversity As An Error Detection Mechanism", submitted to *FTCS 21*.

[17] J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software", *Digest FTCS-16: Proc. 16th Int. Symposium on Fault Tolerant Computing*, Vienna, Austria, July, 1986, pp 165-170.

[18] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986.

[19] N.G. Leveson and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983.

[20] R.J. Lipton, "New Directions in Testing", *Proceedings Interface '90*, East Lansing, MI, May, 1990.

[21] D.L. Lukes, *Differential Equations; Classical to Controlled*, Academic Press, New York. 1982.

[22] D.L. Lukes, "Lie Groups Underlying Fault Avoidance in Dynamical Control Systems", *Proceedings of the 1988 International Conference on Advances in Communication and Control*, Vol.II, pp. 841-848, Baton Rouge, LA, October 1988.

[23] D.R. Miller, "The Role of Statistical Modeling and Inference in Software Quality Assurance", In *Software Certification*, ed. B. de Neumann, Elsevier Applied Science, London, UK, 1989.

[24] L.J. Morell, "A Theory of Fault-Based Testing", *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, August 1990.

[25] P.J. Olver, *Application of Lie Groups to Differential Equations*, Springer-Verlag, New York: 1986.

[26] D.L. Parnas, J van Schouwen and S.P. Kwan, "Evaluation of Safety-Critical Software", *Communications of the ACM*, Vol. 33, No.6, June, 1990.

[27] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[28] F. Saglietti and W. Ehrenberger, "Software Diversity - Some Considerations About Its Benefits And Its Limitations", *Digest of Papers: SAFECOMP '86, 5th International Workshop on Achieving Safe Real-Time Computer Systems*, France, October, 1986.

[29] D.P. Siewiorek and R.S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, MA, USA, 1982.

[30] M.A. Vouk, M.L. Helsabeck, K.C. Tai, and D.F. McAllister, "On Testing of Functionally Equivalent Components of Fault-Tolerant Software", *Proc. COMPSAC 86*, 1986, pp. 414-419.

DISTRIBUTION LIST

1 - 3 National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention: Dr. D. E. Eckhardt, Jr., ISD
M/S 478

4 - 5 * National Aeronautics and Space Administration
Scientific and Technical Information Facility
P. O. Box 8757
Baltimore/Washington International Airport
Baltimore, MD 21240

6 National Aeronautics and Space Administration
Acquisition Division
Langley Research Center
Hampton, VA 23665

Attention: Mr. Richard J. Siebels
Grants Officer, M/S 126

7 - 8 E. H. Pancake, Clark Hall

9 - 10 J. C. Knight, CS

11 A. K. Jones, CS

12 SEAS Preaward Administration Files

*One reproducible copy

JO#3910:ph